University of Hartford – Department of Computing Sciences
CS355 (Computer Networks) – [Final Report]

Vimn't Othello

Wireless Weirdos Raiding Routers

Jacob Hart, Mori Kreider, Riley Wagner, Ryder Raymond

Abstract

An integral aspect of computer networking is the concept of server and client programs and their relations to each other. This programming project explores such a relationship through the creation of an Othello game in the Java language. To complete this task, the group utilized both ends of the network, creating an AI to play against any client in a multi-client environment. The program also utilizes a user interface that the client can interact with directly in order to play the game.

By the end of the project, the group was successful in creating such a program. Everything worked as intended, even if a few aspects of the result weren't optimized completely. If given more time, optimization would be the first thing that would be looked into by the group going into the future. The group would also investigate changing the difficulty of the AI, allowing for player-vs-player games, and making the user interface look nicer.

1.0 Introduction

The goal of this project was to use the Java programming language to develop a networked version of the classic game Othello. It had to revolve around a client-server architecture. The server hosted and managed games, while individual clients could connect to the server to engage in a one-on-one Othello match with it. It was also a necessity for the network to be able to run multiple threads simultaneously. After a match, the server needed to determine a winner before terminating the connection with that specific client.

To accomplish this task, the project utilized Java networking technologies, namely the use of sockets, to establish and maintain these connections between the server and multiple simultaneous clients. As stated before, multithreading was essential due to that specification. Each connection between the server and a client needed to be completely isolated from one another to avoid conflicts.

An additional component of this project was to create a user interface that could convey the game board to the client without confusion. It needed to allow for user input, but it could only accept valid moves. If a client's move was marked as invalid, the interface needed to tell the user such, and it couldn't accept that input.

Developing the AI that would play against the client also required some complexity. It needed to be responsive to the user input, and it had to make strategic inputs based upon what the best moves to make at that moment were. It needed to be consistent between multiple games, as well. Success hinged on the AI's ability to perform as intended while not hampering the rest of the server's logic itself.

In short, the project demonstrated key concepts of computer networking, such as the concurrency of multiple threads and creating an effective user interface design. It also promotes problem-solving and teamwork, as many projects of this sort do, all within the context of a seemingly simple game.

2.0 System Overview

For the server-client architecture and networking component, the server program performs most of the work in terms of getting the changed tiles after placing a move and determining when a player has lost or won the game. The server also contains the AI opponent that automatically responds with a move, which the server sends to the client after a brief delay to allow for viewing what tiles the client's own move has flipped. The server is capable of listening on any port and is able to accept multiple connections simultaneously, keeping track of each game separately. The client must specify the server to connect to, providing either an IP address or hostname (Fully Qualified Domain Name), and the port on which this server is listening. Both server and client automatically disconnect when the game is finished or automatically shut down the connection if the other side encounters an error and quits. The user interface exists on the client program. The user interface displays the board state, instructions, and game messages. It is also used to select a move, which is passed on to the main client object to be sent to the server. Board changes are then passed from the server to the client, which updates the user interface.

3.0 Feature Description

3.1 Key Concepts

Computer networking is an immensely important concept in the modern world. It revolves around two computing devices communicating over wired or wireless networks by sending data between each device. In the case of this system, the main architecture

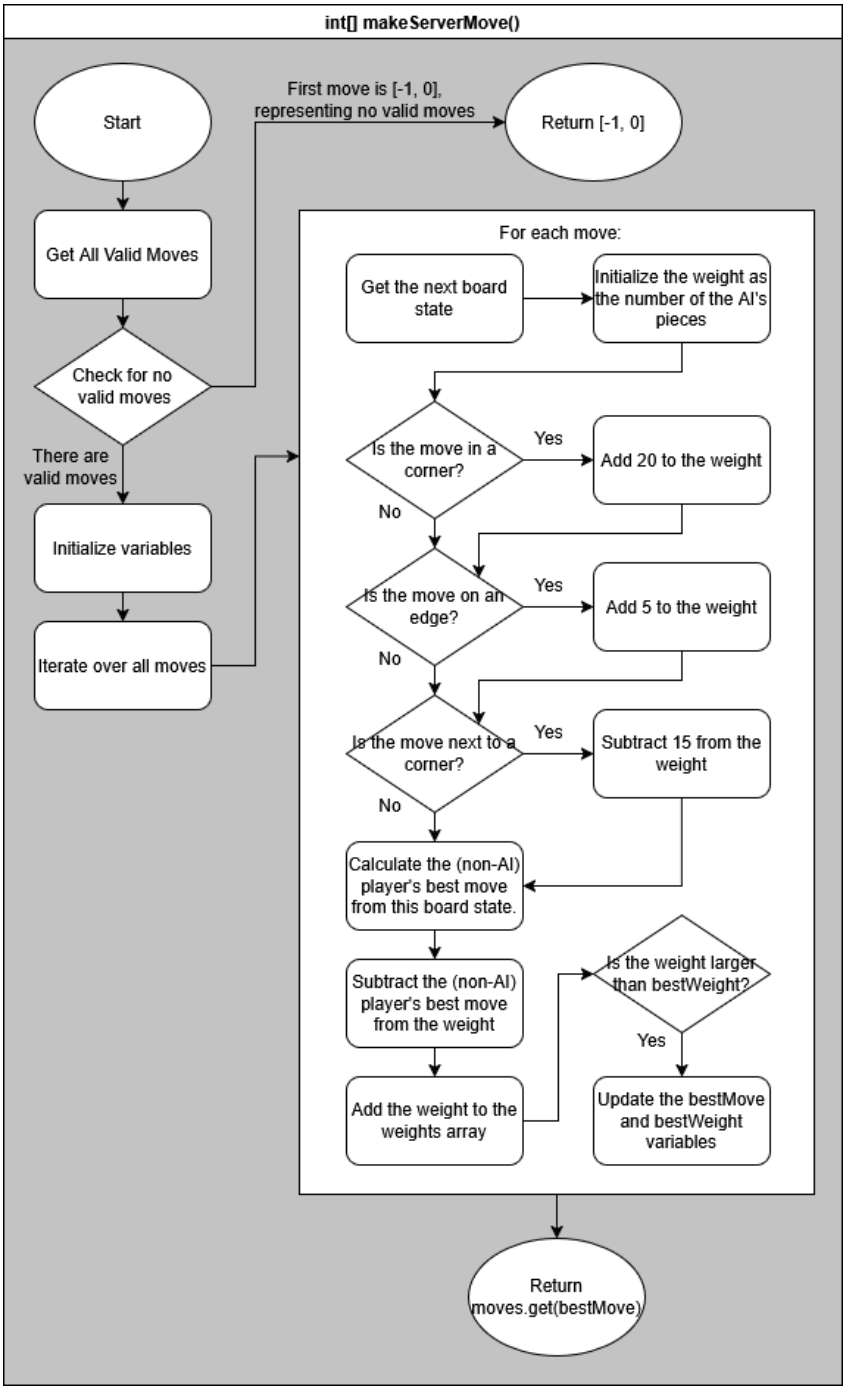
used is the client-server architecture, where one host acts as a server, listening for connections and providing services for clients who connect and request these services, be it printers, file sharing, or computing resources such as the game provided in this system. This system handles networking through the use of sockets, which is a networking concept revolving around an IP (Internet Protocol) address and port number, working similarly to an address of a house and the mailbox number. The IP address and port number combination allows the hosts to locate each other in the network and send data to the other host.

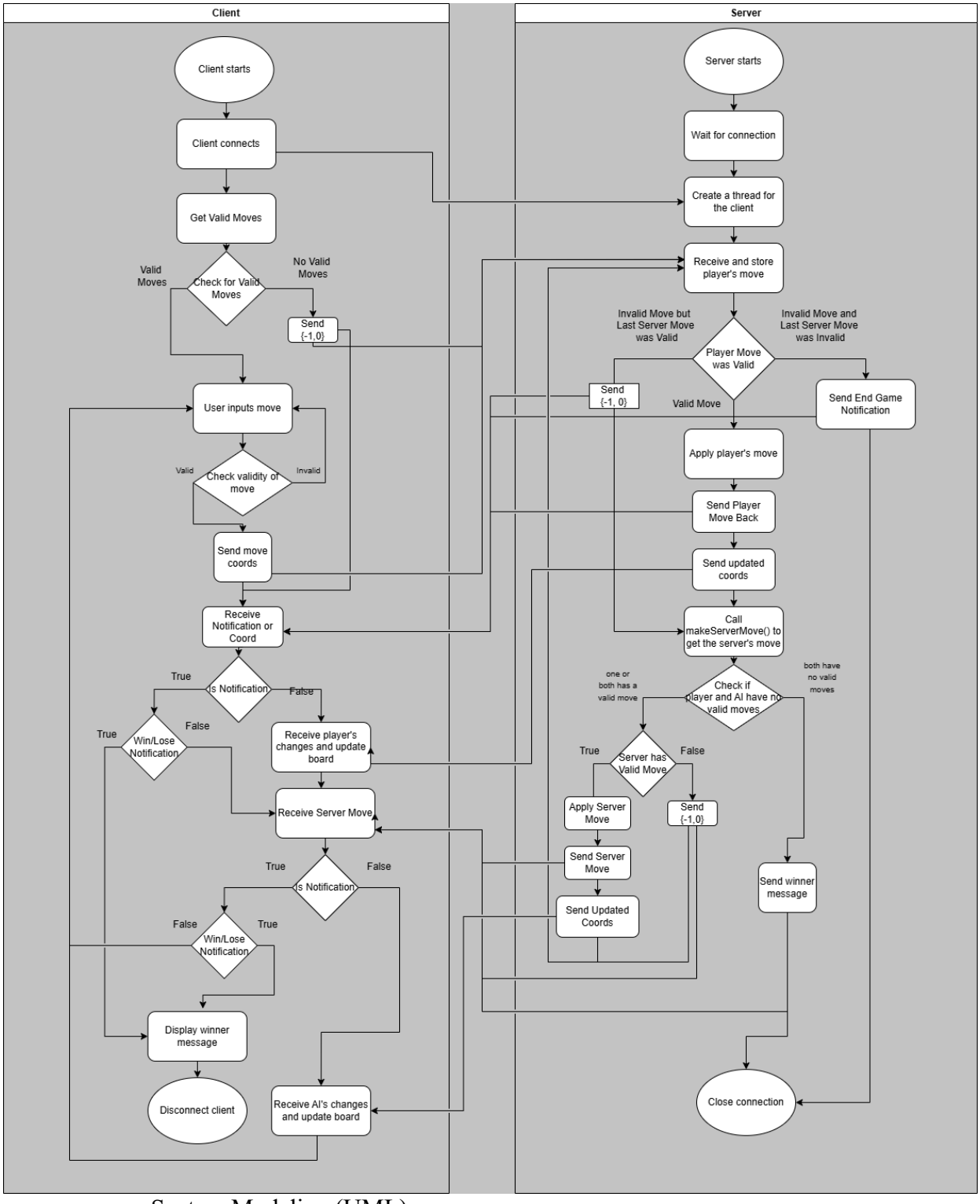
3.2 System Design

a. Design Goal

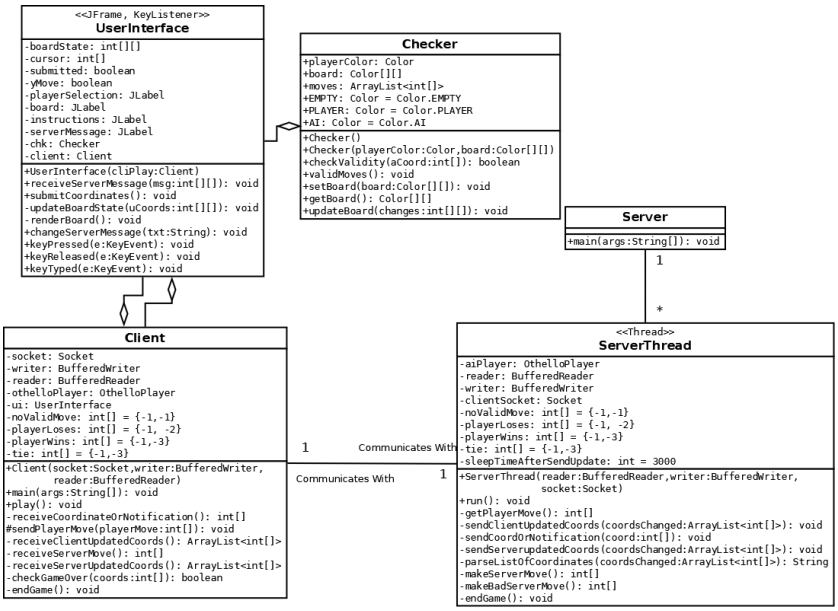
From the user's perspective, the user interface displays instructions, a board, and messages to inform gameplay events such as turns and game results. After submitting a move, a delay is given before the server’s response to allow the player to see how their move affected the board. The user interface also shows the user what specific coordinates they are hovering over at that given moment.

b. Flow Chart


int[] makeServerMove()
Start
First move is [-1, 0], representing no valid moves
Return [-1, 0]
Get All Valid Moves
Check for no valid moves
There are valid moves
Initialize variables
Iterate over all moves
For each move:
Get the next board state
Initialize the weight as the number of the AI's pieces
Is the move in a corner?
Yes
Add 20 to the weight
No
Is the move on an edge?
Yes
Add 5 to the weight
No
Is the move next to a corner?
Yes
Subtract 15 from the weight
No
Calculate the (non-AI) player's best move from this board state.
Subtract the (non-AI) player's best move from the weight
Add the weight to the weights array
Is the weight larger than bestWeight?
Yes
Update the bestMove and bestWeight variables
Return moves.get(bestMove)



c. System Modeling (UML)



4.0 Implementation

4.1 Hardware Components and Experiment

Hardware components used were two laptops, with standard Wi-Fi network interface cards. The laptops were used to test the networking capabilities between two computers, one acting as the server and one acting as the client. The client connected to the server over standard Wi-Fi (WLAN, IEEE 802.11) on the university's Hawknet network, resulting in a successful connection and the ability for the server to serve the remove client alongside a client on the server itself (connected through 'localhost').

The system was not tested over ethernet (IEEE 802.3), although tests were planned and should be as successful as over Wi-Fi. Hardware that would be used for the test are one Ethernet port on one laptop and an Ethernet to USB-A converter for the second laptop.

4.2 Software Solution Development Process

For the user interface, we utilized the Java Swing library for rendering the messages and the board. The board uses HTML tags for line breaks because Java Swift did not like newline characters in text JLabels. We also used Java Swing to detect user keyboard input, which was used to control move submission.

For the networking, Java's standard socket programming and BufferedReaders and Writers from the java.net and java.io packages respectively were used. The algorithm developed was based on the socket programming assignment from class, with the client just needing to know what and when to be reading from the server, and the server containing the majority of the logic. The server performs all logic associated with checking which tiles a move flips, getting the AI player's move, applying it to the board, and sending the list of coordinates that a move has updated on the board to the client whenever either the client or the AI player makes a move. The server also handles checking if a win condition has been met and sending the notification to the client. Tests

were done locally by connecting through 'localhost,' but further tests were conducted over standard Wi-Fi (IEEE 802.11).

To house the Othello board and game functions, there is the OthelloPlayer class which uses ArrayList. An OthelloPlayer object houses a board of colors and 'Color' variables that define the colors as empty, player, or AI. The constructor lets you choose what color the player will be. There are some utility methods like getters and a deepCopyBoard function. The isValidMove method is a static method that takes a 2d array of colors (a board), a coordinate, and a color. It returns true if that coordinate is a valid placement on the board for that color. It does this with a double four loop that iterates through each direction. The x and y variables in the loops go from –1 to 1, in combination representing the eight cardinal and diagonal directions. You can think of x and y forming a vector representing each direction. Then, the algorithm uses another loop to move in that direction for as long as the pieces are the opposite to the specified color. If it reaches another piece with the specified color without going over any empty spaces, then the move is valid. A valid move is one that is placed next to a piece of the opposite color and 'flips' at least one piece by sandwiching it. The getMoves method takes a board and a color and uses isValidMove to check every space to see if that space is a valid move for that color, then it returns a list of valid coordinates or [{-1,0}] if there are no valid moves for that color. There are two place methods, placeOnBoard and place, one returns the resulting board after a piece is placed, and one returns a list of changes that were made to the board after the piece is placed. They use a similar algorithm to isValidMove, but instead of stopping when it finds one direction that would flip a piece, it goes in every direction and finds all opposite pieces that will flip as a result of the move. The bestOutcome method takes a board and a color and returns the best outcome that color could have on that board. It does this by generating all moves for that color and calculating an outcome for each one. The outcome is the number of tiles the move would flip, plus 30 if the move is in a corner, plus 5 if it is on an edge, and minus 15 if it is next to a corner. It does this using the count, isCorner, isEdge, and isNextToCorner methods. Finally, there is the winner method, which returns the color with more pieces on the specified board.

In the server class, there is a method called makeServerMove which houses the AI the server uses to make its moves. The algorithm starts by generating all valid moves the AI can make on the current board, if there are none, it returns {-1,0}. Variables bestMove and bestWeight are initialized, along with an empty int array, the same size as the moves ArrayList. Then, it loops through each move to determine its weight. In the loop, it creates a new board called nextBoard by calling placeOnBoard with the move. It uses this new board to count how many pieces the AI would control and initializes a weight variable as that number. Next, it adds 30 to the weight if that move is in a corner, 5 if it is on an edge, and subtracts 15 if it is next to a corner. The reason it does this is because the corner is the most important spot to capture. The corner can never be captured once a piece is on it and it lets you capture the edges and diagonals. This is also why placing next to a corner is punished, because it lets the enemy more easily get the corner. Then, bestOutcome is called on the nextBoard with the enemy player's color and subtracted from the weight. This punishes moves that let the opponent capture many pieces, or valuable spaces like the corner and edges. It then compares the move's weight to bestWeight and if it is bigger, it sets bestWeight to that weight and sets bestMove to that

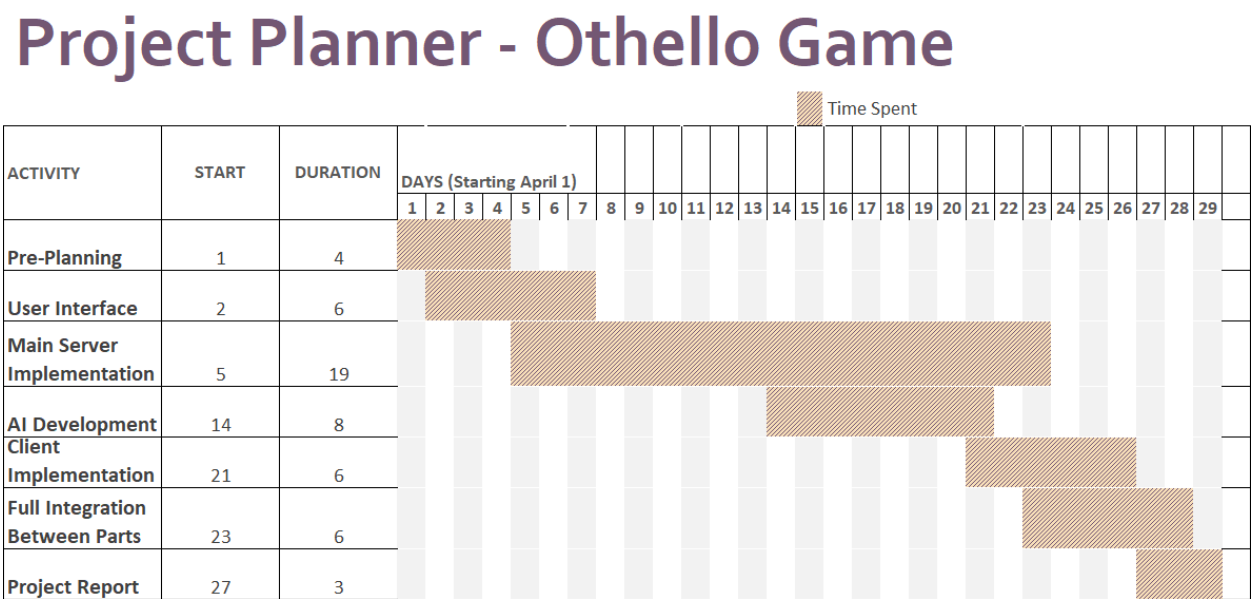
move (represented by the index in the moves ArrayList). After looping through every move, it returns the coordinate of the best move.

4.3 Conclusion and Future Works

The team started the project by first working on the OthelloPlayer and Checker classes, and UI framework. This allowed for testing the main program logic, without worrying about the networking component. Next was the Client and Server architecture, utilizing the other components to perform the majority of the game logic. The project was finished by thoroughly testing the final combined system to ensure the program functions for all end-game scenarios.

For improvements, our team considered a player-versus-player mode where two players could connect to the server or peer-to-peer to play against each other. Our team also considered adding the ability to change the difficulty of the artificial intelligence that the server uses to make moves. One more improvement our team could make would be to improve the graphical user interface to look nicer.

4.4 Tasks



5.0 Key Personnel

Jacob Hart – Hart is an undergraduate student, Computer Science major in the Department of Computing Sciences at the University of Harford. He has completed relevant courses. He has taken relevant courses such as 'Data Structures', and 'Architecture and Assembly Language. He was responsible for the OthelloPlayer and Checker class, as well as some contributions to the Client and Server classes. In Client, he wrote code to deal with the flow of information between the client and server, and code to check and handle a game over. In Server, he was responsible for the AI logic that produces the server's move.

Mori Kreider – Kreider is an undergraduate student, Computer Science major in the Department of Computing Sciences at the University of Harford. She has completed relevant courses in

Java. Kreider was responsible for some of the server-side methods as well as assistance with the client side.

Riley Wagner – Wagner is an undergraduate student, Computer Science major in the Department of Computing Sciences at the University of Harford. He has completed relevant courses in Java and HTML. He was responsible for the user interface.

Ryder Raymond – Raymond is an undergraduate student, Computer Science major in the Department of Computing Sciences at the University of Harford. He has completed Computer Operating Systems in Java, relevant for the experience with socket programming learned through that course. Raymond is also a system administrator, resulting in a solid background with networking concepts. Raymond was responsible for the Server and Client network programming, creating all methods that send player moves and the updated coordinates between the server and client. Raymond completed the major program flow of the server class and its thread class. Raymond also worked with the other team members to develop the client's program flow to be able to read the input from the server and send the client's move at the appropriate time.

6.0 References

[1] Jim Kurose, Keith Ross, "*Computer Networking: A Top-Down Approach*," Eighth Edition, Pearson Education, 2021.

[2] Mark Allen Weiss, "*Data Structures & Problem Solving Using Java,"* Fourth Edition, Pearson Education, 2010.